EXHIBIT D

```
 0 /*    ********************************************************************
 1      File: ciphers.c
 2
 3      SSL Plus: Security Integration Suite(tm)
 4      Version 1.1.1 -- August 11, 1997
 5
 6      Copyright (c)1996, 1997 by Consensus Development Corporation
 7            Copyright (c)1997, 1998 by Aventail Corporation
 8
 9      Portions of this software are based on SSLRef(tm) 3.0, which is
10      Copyright (c)1996 by Netscape Communications Corporation. SSLRef(tm)
11      was developed by Netscape Communications Corporation and Consensus
12      Development Corporation.
13
14      In order to obtain this software, your company must have signed
15      either a PRODUCT EVALUATION LICENSE (a copy of which is included in
16      the file "LICENSE.TXT"), or a PRODUCT DEVELOPMENT LICENSE. These
17      licenses have different limitations regarding how you are allowed to
18      use the software. Before retrieving (or using) this software, you
19      *must* ascertain which of these licenses your company currently
20      holds. Then, by retrieving (or using) this software you agree to
21      abide by the particular terms of that license. If you do not agree
22      to abide by the particular terms of that license, than you must
23      immediately delete this software. If your company does not have a
24      signed license of either kind, then you must either contact
25      Consensus Development and execute a valid license before retrieving
26      (or using) this software, or immediately delete this software.
27
28      ********************************************************************
29
30      File: ciphers.c    Data structures for handling supported ciphers
31
32      Contains a table mapping cipherSuite values to the ciphers, MAC
33      algorithms, key exchange procedures and so on that are used for that
34      algorithm, in order of preference.
35
36      ******************************************************************** */
37
38 #ifndef _CRYPTYPE_H_
39 #include <cryptype.h>
40 #endif
41
42 #ifndef _SSLCTX_H_
43 #include <sslctx.h>
44 #endif
45
46 #include <string.h>
47
48 extern SSLSymmetricCipher SSLCipherNull;
49 extern SSLSymmetricCipher SSLCipherDES_CBC;
50 extern SSLSymmetricCipher SSLCipherDES40_CBC;
51 extern SSLSymmetricCipher SSLCipherRC4_40;
52 extern SSLSymmetricCipher SSLCipherRC4_56;
53 extern SSLSymmetricCipher SSLCipherRC4_128;
54 extern SSLSymmetricCipher SSLCipher3DES_CBC;
55
56 /* Even if we don't support NULL_WITH_NULL_NULL for transport, we need a reference for startup
      */
57 SSLCipherSpec SSL_NULL_WITH_NULL_NULL_CipherSpec =
58 {    SSL_NULL_WITH_NULL_NULL,
59      Exportable,
60      SSL_NULL_auth,
61      &SSLHashNullOpt,
62      &SSLCipherNull
63 };
64
65 /* Disable non-exportable cipher suites to build an export only library */
66 #ifndef ENABLE_NONEXPORT_CIPHERS
67 #define ENABLE_NONEXPORT_CIPHERS 1
68 #endif
69
```

```
70  /* Disable exportable cipher suites to build a strong crypto only library */
71  #ifndef ENABLE_EXPORT_CIPHERS
72  #define ENABLE_EXPORT_CIPHERS 1
73  #endif
74
75  /* Reenable DH-anon only if you know you want to use Diffie-Hellman cipher suites:
76      Enabling DH-anon leaves you open to a man-in-the-middle attack which can degrade your
77       security to this level. */
78  #ifndef ENABLE_DH_ANON
79  #define ENABLE_DH_ANON 0
80  #endif
81
82  /* Reenable NULL encryption cipher suites only if you know for a fact you want to support
83      unencrypted sessions. Unencrypted sessions do not provide data privacy and may be more
84      vulnerable to attack than encrypted sessions. */
85  #ifndef ENABLE_NULL_CIPHERS
86  #define ENABLE_NULL_CIPHERS 0
87  #endif
88
89  #ifdef VIRGIN_SSLPLUS
90  /* Order by preference */
91  SSLCipherSpec KnownCipherSpecs[] =
92  {
93  #if ENABLE_NONEXPORT_CIPHERS
94      {   SSL_RSA_WITH_3DES_EDE_CBC_SHA, NotExportable, SSL_RSA, &SSLHashSHA1, &SSLCipher3DES_CBC
      },
95      {   SSL_RSA_WITH_RC4_128_SHA, NotExportable, SSL_RSA, &SSLHashSHA1, &SSLCipherRC4_128 },
96      {   SSL_RSA_WITH_RC4_128_MD5, NotExportable, SSL_RSA, &SSLHashMD5, &SSLCipherRC4_128 },
97      {   SSL_RSA_WITH_DES_CBC_SHA, NotExportable, SSL_RSA, &SSLHashSHA1, &SSLCipherDES_CBC },
98  #endif
99  #if ENABLE_EXPORT_CIPHERS
100     {   SSL_RSA_EXPORT_WITH_RC4_40_MD5, Exportable, SSL_RSA_EXPORT, &SSLHashMD5,
      &SSLCipherRC4_40 },
101     {   SSL_RSA_EXPORT_WITH_DES40_CBC_SHA, Exportable, SSL_RSA_EXPORT, &SSLHashSHA1,
      &SSLCipherDES40_CBC },
102 #endif
103 #if ENABLE_DH_ANON && ENABLE_NONEXPORT_CIPHERS
104     {   SSL_DH_anon_WITH_3DES_EDE_CBC_SHA, NotExportable, SSL_DH_anon, &SSLHashSHA1,
      &SSLCipher3DES_CBC },
105     {   SSL_DH_anon_WITH_RC4_128_MD5, NotExportable, SSL_DH_anon, &SSLHashMD5,
      &SSLCipherRC4_128 },
106     {   SSL_DH_anon_WITH_DES_CBC_SHA, NotExportable, SSL_DH_anon, &SSLHashSHA1,
      &SSLCipherDES_CBC },
107 #endif
108 #if ENABLE_NULL_CIPHERS && ENABLE_EXPORT_CIPHERS
109     {   SSL_RSA_WITH_NULL_SHA, Exportable, SSL_RSA, &SSLHashSHA1, &SSLCipherNull },
110     {   SSL_RSA_WITH_NULL_MD5, Exportable, SSL_RSA, &SSLHashMD5, &SSLCipherNull }
111 #endif
112 };
113
114 int CipherSpecCount = sizeof(KnownCipherSpecs) / sizeof(SSLCipherSpec);
115 #endif /* VIRGIN_SSLPLUS */
116
117 SSLErr
118 FindCipherSpec(SSLContext *ctx, uint16 specID, SSLCipherSpec* *spec)
119 {
120     int i;
121     uint32 mask;
122
123     *spec = 0;
124     for (i = 0; i < CipherSpecCount; i++)
125     {
126         if (KnownCipherSpecs[i].cipherSpec == specID)
127         {
128             mask = (uint32) 1;
129             mask <<= i;
130             if(ctx->cipherspecs & mask)
131             {
132                 *spec = &KnownCipherSpecs[i];
133                 break;
134             }
```

```
135            }
136        }
137
138      if (*spec == 0)          /* Not found */
139          return SSLNegotiationErr;
140      return SSLNoErr;
141 }
142
143 SSLErr SSLDESInit(uint8 *key, uint8* iv, void **cipherRef, SSLContext *ctx);
144 SSLErr SSLDESEncrypt(SSLBuffer src, SSLBuffer dest, SSLBuffer *iv, void *cipherRef, SSLContext
        *ctx);
145 SSLErr SSLDESDecrypt(SSLBuffer src, SSLBuffer dest, SSLBuffer *iv, void *cipherRef, SSLContext
        *ctx);
146 SSLErr SSLDESFinish(void *cipherRef, SSLContext *ctx);
147 SSLErr SSLDESExport(void *cipherRef, SSLContext *ctx, SSLBuffer *blob);
148 SSLErr SSLDESImport(void **cipherRef, SSLContext *ctx, SSLBuffer *blob);
149
150 SSLSymmetricCipher SSLCipherDES_CBC = {
151      8,         /* Key size in bytes */
152      8,         /* Secret key size = 64 bits */
153      8,         /* IV size */
154      8,         /* Block size */
155      SSLDESInit,
156      SSLDESEncrypt,
157      SSLDESDecrypt,
158      SSLDESFinish,
159      SSLDESExport,
160      SSLDESImport
161 };
162
163 SSLSymmetricCipher SSLCipherDES40_CBC = {
164      8,         /* Key size in bytes */
165      5,         /* Secret key size = 40 bits */
166      8,         /* IV size */
167      8,         /* Block size */
168      SSLDESInit,
169      SSLDESEncrypt,
170      SSLDESDecrypt,
171      SSLDESFinish
172 };
173
174 typedef struct _DESState
175 {
176      unsigned char key[24]; /* work for 3DES and DES both */
177      unsigned char iv[8];
178      int reading; /* do we really need this? */
179      B_ALGORITHM_OBJ des;
180 } DESState;
181
182 SSLErr
183 SSLDESInit(uint8 *key, uint8* iv, void **cipherRef, SSLContext *ctx)
184 {
185      SSLBuffer                        desState;
186      B_ALGORITHM_OBJ          *des;
187      static B_ALGORITHM_METHOD  *chooser[] = { &AM_DES_CBC_ENCRYPT, &AM_DES_CBC_DECRYPT, 0 };
188      B_KEY_OBJ                  desKey;
189      ITEM                       keyData;
190      SSLErr                     err;
191      int                        rsaErr;
192      DESState *s;
193
194      if ((err = SSLAllocBuffer(&desState, sizeof(DESState), &ctx->sysCtx)) != 0)
195          return err;
196      s = (DESState *)desState.data;
197
198      memcpy(s->key, key, 8);
199      memcpy(s->iv, iv, 8);
200
201      if ((rsaErr = B_CreateAlgorithmObject(&(s->des))) != 0)
202          return SSLUnknownErr;
```

```
203        if ((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_CBC_IV8, iv)) != 0)
204            return SSLUnknownErr;
205        if ((rsaErr = B_CreateKeyObject(&desKey)) != 0)
206            return SSLUnknownErr;
207        keyData.data = key;
208        keyData.len = 8;
209        if ((rsaErr = B_SetKeyInfo(desKey, KI_DES8, key)) != 0)
210        {   B_DestroyKeyObject(&desKey);
211            return SSLUnknownErr;
212        }
213        if (cipherRef == (void**)&(ctx->writePending.symCipherState))
214        {
215            s->reading = 0;
216            if ((rsaErr = B_EncryptInit(*des, desKey, chooser, &ctx->sysCtx.yield)) != 0)
217            {
218                    B_DestroyKeyObject(&desKey);
219                return SSLUnknownErr;
220            }
221        }
222        else if (cipherRef == (void**)&(ctx->readPending.symCipherState))
223        {
224            s->reading = 1;
225            if ((rsaErr = B_DecryptInit(*des, desKey, chooser, &ctx->sysCtx.yield)) != 0)
226            {
227                    B_DestroyKeyObject(&desKey);
228                return SSLUnknownErr;
229            }
230        }
231        else
232            ASSERTMSG("Couldn't determine read/writeness");
233
234        B_DestroyKeyObject(&desKey);
235        *cipherRef = (void*)s;
236        return SSLNoErr;
237 }
238
239 SSLErr
240 SSLDESEncrypt(SSLBuffer src, SSLBuffer dest, SSLBuffer *iv, void *cipherRef, SSLContext *ctx)
241 {
242    DESState *s = (DESState *) cipherRef;
243    void *subCipherRef = NULL;
244    int        rsaErr;
245    unsigned int        outputLen;
246    SSLBuffer        temp;
247    SSLErr        err;
248
249    if(cipherRef == NULL)
250            return SSLUnknownErr;
251
252    if(iv != NULL)
253    {
254            if((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_CBC_IV8,
255                                                            (POINTER) iv->data)) !=
        SSLNoErr)
256                    return err;
257    }
258    else
259    {
260            if((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_CBC_IV8, s->iv)) != SSLNoErr)
261                    return err;
262    }
263
264    ASSERT(src.length == dest.length);
265    ASSERT(src.length % 8 == 0);
266
267    if (src.data == dest.data)
268 /* BSAFE won't let you encrypt in place */
269    {   if (ERR(err = SSLAllocBuffer(&temp, src.length, &ctx->sysCtx)) != 0)
270            return err;
271        memcpy(temp.data, src.data, (size_t) src.length);
272    }
```

```
273      else
274        temp = src;
275
276      if ((rsaErr = B_EncryptUpdate(s->des, dest.data, &outputLen,
277                          (unsigned int) dest.length, temp.data,
278                          (unsigned int) temp.length,
279                    (B_ALGORITHM_OBJ) 0, &ctx->sysCtx.yield)) != 0)
280      {   if (src.data == dest.data)
281              SSLFreeBuffer(&temp, &ctx->sysCtx);
282          return SSLUnknownErr;
283      }
284
285      ASSERT(outputLen == src.length);
286
287      if (src.data == dest.data)
288          SSLFreeBuffer(&temp, &ctx->sysCtx);
289
290      if (outputLen != src.length)
291          return SSLUnknownErr;
292
293      /* if not doing SSLoppy, save the IV for next time... */
294      if(iv == NULL)
295      {
296          unsigned char *buf;
297
298          if((rsaErr = B_GetAlgorithmInfo((POINTER *) &buf, s->des,
299                                                          AI_DES_CBC_IV8))
300              != SSLNoErr)
301                  return err;
302
303          memcpy(s->iv, buf, sizeof(s->iv));
304      }
305
306 /* memcpy(s->iv, dest.data + dest.length - 8, 8); */
307
308      return SSLNoErr;
309 }
310
311 SSLErr
312 SSLDESDecrypt(SSLBuffer src, SSLBuffer dest, SSLBuffer *iv, void *cipherRef, SSLContext *ctx)
313 {
314      DESState *s = (DESState *) cipherRef;
315      int              rsaErr;
316      unsigned int        outputLen;
317      SSLBuffer           temp;
318      SSLErr           err;
319
320      if(cipherRef == NULL)
321              return SSLUnknownErr;
322
323      if(iv != NULL)
324      {
325              if((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_CBC_IV8, (POINTER) iv->data))
326                  != SSLNoErr)
327                      return err;
328      }
329      else
330      {
331              if((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_CBC_IV8, s->iv)) != SSLNoErr)
332                      return err;
333      }
334
335      ASSERT(src.length == dest.length);
336      ASSERT(src.length % 8 == 0);
337
338 /* memcpy(s->iv, src.data + src.length - 8, 8); */
339
340      if (src.data == dest.data)
341 /* BSAFE won't let you encrypt in place */
342      {   if (ERR(err = SSLAllocBuffer(&temp, src.length, &ctx->sysCtx)) != 0)
343              return err;
```

```
344             memcpy(temp.data, src.data, (size_t) src.length);
345         }
346         else
347             temp = src;
348
349         if ((rsaErr = B_DecryptUpdate(s->des, dest.data, &outputLen,
350                             (unsigned int) dest.length, temp.data,
351                             (unsigned int) temp.length,
352                     (B_ALGORITHM_OBJ) 0, &ctx->sysCtx.yield)) != 0)
353         {   if (src.data == dest.data)
354                 SSLFreeBuffer(&temp, &ctx->sysCtx);
355             return SSLUnknownErr;
356         }
357
358         ASSERT(outputLen == src.length);
359
360         if (src.data == dest.data)
361             SSLFreeBuffer(&temp, &ctx->sysCtx);
362
363         if (outputLen != src.length)
364             return SSLUnknownErr;
365
366     /* if not doing SSLoppy, save the IV for next time... */
367     if(iv == NULL)
368     {
369             unsigned char *buf;
370
371             if((rsaErr = B_GetAlgorithmInfo((POINTER *) &buf, s->des,
372                                                                             AI_DES_CBC_IV8))
373                 != SSLNoErr)
374                     return err;
375             memcpy(s->iv, buf, sizeof(s->iv));
376     }
377
378     return SSLNoErr;
379 }
380
381 SSLErr
382 SSLDESFinish(void *cipherRef, SSLContext *ctx)
383 {
384     DESState *s = (DESState *) cipherRef;
385     SSLBuffer           desState;
386     SSLErr          err;
387
388     if(cipherRef == NULL)
389             return SSLUnknownErr;
390
391     B_DestroyAlgorithmObject(&(s->des));
392
393     memset(cipherRef, 0, sizeof(DESState));
394     desState.data = (unsigned char*)cipherRef;
395     desState.length = sizeof(DESState);
396
397     err = SSLFreeBuffer(&desState, &ctx->sysCtx);
398     return err;
399 }
400
401 SSLErr SSLDESExport(void *cipherRef, SSLContext *ctx, SSLBuffer *blob)
402 {
403     DESState *s = (DESState *) cipherRef;
404
405     if(cipherRef == NULL)
406             return SSLUnknownErr;
407
408     if(blob->length < (8 + 8))
409             return SSLMemoryErr;
410
411     memcpy(blob->data, s->key, 8);
412     memcpy(blob->data + 8, s->iv, 8);
413 /* memcpy(blob->data + 16, &(s->reading), sizeof(int)); */
414     blob->length = 16;
```

```
415
416     return SSLNoErr;
417 }
418
419 SSLErr SSLDESImport(void **cipherRef, SSLContext *ctx, SSLBuffer *blob)
420 {
421     unsigned char *key, *iv;
422
423     if(blob == NULL)
424             return SSLUnknownErr;
425     if(blob->length < 16)
426             return SSLUnknownErr;
427
428     key = blob->data;
429     iv = blob->data + 8;
430
431     return SSLDESInit(key, iv, cipherRef, ctx);
432 }
433
434
435 SSLErr SSL3DESInit(uint8 *key, uint8* iv, void **cipherRef, SSLContext *ctx);
436 SSLErr SSL3DESEncrypt(SSLBuffer src, SSLBuffer dest, SSLBuffer *iv, void *cipherRef, SSLContext
        *ctx);
437 SSLErr SSL3DESDecrypt(SSLBuffer src, SSLBuffer dest, SSLBuffer *iv, void *cipherRef, SSLContext
        *ctx);
438 SSLErr SSL3DESFinish(void *cipherRef, SSLContext *ctx);
439 SSLErr SSL3DESExport(void *cipherRef, SSLContext *ctx, SSLBuffer *blob);
440 SSLErr SSL3DESImport(void **cipherRef, SSLContext *ctx, SSLBuffer *blob);
441
442 SSLSymmetricCipher SSLCipher3DES_CBC = {
443      24,      /* Key size in bytes */
444      24,      /* Secret key size = 192 bits */
445      8,       /* IV size */
446      8,       /* Block size */
447      SSL3DESInit,
448      SSL3DESEncrypt,
449      SSL3DESDecrypt,
450      SSL3DESFinish,
451    SSL3DESExport,
452    SSL3DESImport
453 };
454
455 SSLErr
456 SSL3DESInit(uint8 *key, uint8* iv, void **cipherRef, SSLContext *ctx)
457 {
458    SSLBuffer                    desState;
459    DESState *s;
460      static B_ALGORITHM_METHOD    *chooser[] = { &AM_DES_EDE3_CBC_ENCRYPT,
461
        &AM_DES_EDE3_CBC_DECRYPT, 0 };
462      B_KEY_OBJ                    desKey;
463      ITEM                         keyData;
464      SSLErr                       err;
465      int                          rsaErr;
466
467      if ((err = SSLAllocBuffer(&desState, sizeof(DESState), &ctx->sysCtx)) != 0)
468          return err;
469      s = (DESState *)desState.data;
470      if ((rsaErr = B_CreateAlgorithmObject(&(s->des))) != 0)
471          return SSLUnknownErr;
472      if ((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_EDE3_CBC_IV8, iv)) != 0)
473          return SSLUnknownErr;
474    memcpy(s->iv, iv, 8);
475
476      if ((rsaErr = B_CreateKeyObject(&desKey)) != 0)
477          return SSLUnknownErr;
478      keyData.data = key;
479      keyData.len = 24;
480      if ((rsaErr = B_SetKeyInfo(desKey, KI_24Byte, key)) != 0)
481      {
482              B_DestroyKeyObject(&desKey);
```

```
483           return SSLUnknownErr;
484      }
485    memcpy(s->key, key, 24);
486
487      if (cipherRef == (void**)&(ctx->writePending.symCipherState))
488      {
489          if ((rsaErr = B_EncryptInit(s->des, desKey, chooser,
490                                                    &ctx->sysCtx.yield)) != 0)
491          {
492              B_DestroyKeyObject(&desKey);
493            return SSLUnknownErr;
494          }
495      }
496      else if (cipherRef == (void**)&(ctx->readPending.symCipherState))
497      {
498          if ((rsaErr = B_DecryptInit(s->des, desKey, chooser,
499                                                    &ctx->sysCtx.yield)) != 0)
500          {
501              B_DestroyKeyObject(&desKey);
502            return SSLUnknownErr;
503          }
504      }
505      else
506          ASSERTMSG("Couldn't determine read/writeness");
507
508      B_DestroyKeyObject(&desKey);
509      *cipherRef = (void*)desState.data;
510      return SSLNoErr;
511 }
512
513 SSLErr
514 SSL3DESEncrypt(SSLBuffer src, SSLBuffer dest, SSLBuffer *iv, void *cipherRef, SSLContext *ctx)
515 {
516    DESState *s =(DESState *) cipherRef;
517    int            rsaErr;
518    unsigned int   outputLen;
519    SSLBuffer      temp;
520    SSLErr         err;
521
522    ASSERT(src.length == dest.length);
523    ASSERT(src.length % 8 == 0);
524    if(cipherRef == NULL)
525          return SSLUnknownErr;
526
527    if(iv != NULL)
528    {
529          if((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_EDE3_CBC_IV8,
530                                                    (POINTER) iv->data)) !=
    SSLNoErr)
531              return err;
532    }
533    else
534    {
535          if((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_EDE3_CBC_IV8, s->iv)) != SSLNoErr)
536              return err;
537    }
538
539
540    if (src.data == dest.data)
541 /* BSAFE won't let you encrypt in place */
542    {   if (ERR(err = SSLAllocBuffer(&temp, src.length, &ctx->sysCtx)) != 0)
543            return err;
544        memcpy(temp.data, src.data, (size_t) src.length);
545    }
546    else
547        temp = src;
548
549    if ((rsaErr = B_EncryptUpdate(s->des, dest.data, &outputLen,
550                          (unsigned int) dest.length, temp.data,
551                          (unsigned int) temp.length,
552                      (B_ALGORITHM_OBJ) 0, &ctx->sysCtx.yield)) != 0)
```

```
553        {   if (src.data == dest.data)
554                SSLFreeBuffer(&temp, &ctx->sysCtx);
555            return SSLUnknownErr;
556        }
557
558        ASSERT(outputLen == src.length);
559
560        if (src.data == dest.data)
561            SSLFreeBuffer(&temp, &ctx->sysCtx);
562
563        if (outputLen != src.length)
564            return SSLUnknownErr;
565
566    if(iv == NULL)
567        {
568            unsigned char *buf;
569
570            if((rsaErr = B_GetAlgorithmInfo((POINTER *) &buf, s->des,
571                                                                    AI_DES_EDE3_CBC_IV8))
572                != SSLNoErr)
573                    return err;
574            memcpy(s->iv, buf, sizeof(s->iv));
575        }
576
577 /* memcpy(s->iv, dest.data + dest.length - 8, 8); */
578
579        return SSLNoErr;
580 }
581
582 SSLErr
583 SSL3DESDecrypt(SSLBuffer src, SSLBuffer dest, SSLBuffer *iv, void *cipherRef, SSLContext *ctx)
584 {
585    DESState *s = (DESState *) cipherRef;
586    int              rsaErr;
587    unsigned int         outputLen;
588    SSLBuffer            temp;
589    SSLErr           err;
590
591    ASSERT(src.length == dest.length);
592    ASSERT(src.length % 8 == 0);
593    if(cipherRef == NULL)
594            return SSLNoErr;
595
596.   if(iv != NULL)
597        {
598            if((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_EDE3_CBC_IV8,
599                                                                (POINTER) iv->data)) !=
        SSLNoErr)
600                    return err;
601        }
602    else
603        {
604            if((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_EDE3_CBC_IV8, s->iv)) != SSLNoErr)
605                    return err;
606        }
607
608 /* memcpy(s->iv, src.data + src.length - 8, 8);  */
609
610     if (src.data == dest.data)
611 /* BSAFE won't let you encrypt in place */
612        {   if (ERR(err = SSLAllocBuffer(&temp, src.length, &ctx->sysCtx)) != 0)
613                return err;
614            memcpy(temp.data, src.data, (size_t) src.length);
615        }
616    else
617            temp = src;
618
619        if ((rsaErr = B_DecryptUpdate(s->des, dest.data, &outputLen,
620                            (unsigned int) dest.length, temp.data,
621                            (unsigned int) temp.length,
622                        (B_ALGORITHM_OBJ) 0, &ctx->sysCtx.yield)) != 0)
```

```
623        {   if (src.data == dest.data)
624               SSLFreeBuffer(&temp, &ctx->sysCtx);
625           return SSLUnknownErr;
626        }
627
628     if(iv == NULL)
629     {
630            unsigned char *buf;
631
632            if((rsaErr = B_GetAlgorithmInfo((POINTER *) &buf, s->des,
633                                                                AI_DES_EDE3_CBC_IV8)) !=
        SSLNoErr)
634                    return err;
635            memcpy(s->iv, buf, sizeof(s->iv));
636        }
637
638      ASSERT(outputLen == src.length);
639
640      if (src.data == dest.data)
641          SSLFreeBuffer(&temp, &ctx->sysCtx);
642
643      if (outputLen != src.length)
644          return SSLUnknownErr;
645
646      return SSLNoErr;
647 }
648
649 SSLErr
650 SSL3DESFinish(void *cipherRef, SSLContext *ctx)
651 {
652     DESState *s = (DESState *) cipherRef;
653       SSLBuffer            desState;
654       SSLErr           err;
655
656     if(cipherRef == NULL)
657             return SSLUnknownErr;
658
659     B_DestroyAlgorithmObject(&(s->des));
660
661     memset(cipherRef, 0, sizeof(DESState));
662       desState.data = (unsigned char*)cipherRef;
663       desState.length = sizeof(DESState);
664       err = SSLFreeBuffer(&desState, &ctx->sysCtx);
665       return err;
666 }
667
668 SSLErr SSL3DESExport(void *cipherRef, SSLContext *ctx, SSLBuffer *blob)
669 {
670     DESState *s = (DESState *) cipherRef;
671
672     if(cipherRef == NULL)
673             return SSLUnknownErr;
674
675     if(blob->length < (24 + 8))
676             return SSLMemoryErr;
677
678     memcpy(blob->data, s->key, 24);
679     memcpy(blob->data + 24, s->iv, 8);
680     blob->length = 32;
681
682     return SSLNoErr;
683 }
684
685 SSLErr SSL3DESImport(void **cipherRef, SSLContext *ctx, SSLBuffer *blob)
686 {
687   unsigned char *key, *iv;
688
689     if(blob == NULL)
690             return SSLUnknownErr;
691     if(blob->length < 32)
692             return SSLUnknownErr;
```

```
693
694     key = blob->data;
695     iv = blob->data + 24;
696
697     return SSL3DESInit(key, iv, cipherRef, ctx);
698 }
```